

Kamino Conditional Liquidity

Security Assessment

March 16th, 2026 — Prepared by OtterSec

Tamta Topuria tamta@osec.io

Artem Astakhov aikasimirov@osec.io

Table of Contents

Executive Summary **2**

Overview 2

Key Findings 2

Scope **3**

Findings **4**

Vulnerabilities **5**

OS-KCL-ADV-00 | Stale Invalid Ticket Queue Accounting 6

OS-KCL-ADV-01 | Inability of Callback Accounts to Accept Klend ID 7

Appendices

Vulnerability Rating Scale **8**

Procedure **9**

© 2026 Otter Audits LLC. All Rights Reserved.

01 — Executive Summary

Overview

Kamino Finance engaged OtterSec to assess the `conditional-liquidity` feature. This assessment was conducted between February 7th and March 9th, 2026. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 2 findings throughout this audit engagement.

In particular, we identified a vulnerability where the variable that tracks queued reserve exits is only cleared on the successful-withdraw callback path, so if a withdraw ticket is marked invalid and resolved through recovery instead, Kvault may retain stale queued-cToken accounting (OS-KCL-ADV-00). Furthermore, utilizing Option<AccountInfo> renders Klend's own program ID unrepresentable in optional callback fields. Because Anchor interprets that public key as the sentinel for None, it results in future callback paths that legitimately require crate::ID there to silently deserialize it as None (OS-KCL-ADV-01).

© 2026 Otter Audits LLC. All Rights Reserved.

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/Kamino-Finance/klend-private. Audits were performed on the following PRs:

1. PR#627
2. PR#628
3. PR#632
4. PR#662
5. PR#704

A brief description of the program is as follows:

Name	Description
conditional-liquidity	It introduces a queued-withdraw, ticket-based liquidity flow in which withdrawals can be redeemed asynchronously as reserve liquidity becomes available, rather than requiring immediate liquidity. It also adds progress tracking and callback integration, enabling external systems such as Kvault to synchronize their accounting when queued collateral is redeemed or otherwise resolved.

© 2026 Otter Audits LLC. All Rights Reserved.

03 — Findings

Overall, we reported 2 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	0
LOW	2
INFO	0

© 2026 Otter Audits LLC. All Rights Reserved.

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-KCL-ADV-00	LOW	RESOLVED	Currently, `klend_queued_ctokens` is only cleared on the successful-withdraw callback path, so if a withdraw ticket is marked invalid and resolved through recovery instead, `Kvault` may retain stale queued-`cToken` accounting.
OS-KCL-ADV-01	LOW	RESOLVED	Utilizing `Option<AccountInfo>` renders `Klend`'s own program ID unrepresentable in optional callback fields because Anchor interprets that public key as the sentinel for `None`, resulting in future callback paths that legitimately require `crate::ID` there to silently deserialize it as `None`.

© 2026 Otter Audits LLC. All Rights Reserved.

Stale Invalid Ticket Queue Accounting LOW OS-KCL-ADV-00

Description

`Kvault` tracks queued reserve exits in `klend_queued_ctokens` assuming that queued-withdraw progress will always be finalized through the normal success callback path. Under that assumption, `Kvault` decrements `klend_queued_ctokens` only when the callback reports `QueuedLiquidityWithdrawn`, the burned `cToken` amount as the quantity that has left the queued state. This becomes incorrect if a withdraw ticket is later marked invalid. In that case, the queued `cTokens` are no longer pending withdrawal in the original sense, but the normal success callback is never executed.

>_ *programs/kvault/src/operations/vault_operations.rs* RUST

```rust
pub fn update_klend_queue_accounting(
    vault: &mut VaultState,
    reserve_address: Pubkey,
    burnt_ctokens: u64,
    received_liquidity: u64,
) -> Result<()> {
    let allocation = vault.allocation_for_reserve_mut(&reserve_address)?;
    allocation.klend_queued_ctokens -= burnt_ctokens;
    allocation.ctoken_allocation -= burnt_ctokens;
    vault.token_available += received_liquidity;
    Ok(())
}
```

Instead, recovery occurs through `handler_recover_invalid_ticket_collateral`. As a result, `Kvault` may retain stale queued-`cToken` accounting even after the ticket has been invalidated and the collateral has been recovered through a different path. This inconsistency is particularly problematic because `Kvault` separately tracks both `ctoken_allocation` and `klend_queued_ctokens`, and these fields represent different accounting dimensions. Invalidating a ticket should not be triggerable for Kvault-created tickets. However, future callback types might legitimately produce invalid withdraw tickets, in which case Klend would need to handle this scenario.

Remediation

Extend the `Kvault` callback payload with an explicit status flag indicating whether the ticket was redeemed successfully or instead marked invalid, and implement corresponding invalid-ticket handling logic within the `kvault` callback instruction.

Patch

This issue was acknowledged by the team.

© 2026 Otter Audits LLC. All Rights Reserved.

Inability of Callback Accounts to Accept Klend ID LOW OS-KCL-ADV-01

Description

Anchor's `Option<T>` account parsing utilizes a special sentinel rule, if the next account's key equals the current program ID, Anchor consumes that account and returns `None` instead of `Some(AccountInfo)`.

>_ *anchor/lang/src/accounts/option.rs* RUST

```rust
fn try_accounts(
    program_id: &Pubkey,
    accounts: &mut &'info [AccountInfo<'info>],
    ix_data: &[u8],
    bumps: &mut B,
    reallocs: &mut BTreeSet<Pubkey>,
) -> Result<Self> {
   [...]
    // If there are enough accounts, it will check the program_id and return
    // None if it matches, popping the first account off the accounts vec.
    if accounts[0].key == program_id {
        *accounts = &accounts[1..];
        Ok(None)
    } else {
        // If the program_id doesn't equal the account key, we default to
        // the try_accounts implementation for the inner type and then wrap that with
        // Some. This should handle all possible valid cases.
        T::try_accounts(program_id, accounts, ix_data, bumps, reallocs).map(Some)
    }
}
```

Since `progress_callback_program`, `progress_callback_custom_account_0`, and `progress_callback_custom_account_1` are all modeled as `Option<AccountInfo<'info>>`, if any of those callback account slots ever legitimately require to contain the `Klend` program id itself, Anchor will not surface that account to the instruction logic. It will instead store the field as `None` in the `Context`. This is not an issue currently if no callback path needs crate::ID, however it creates a hidden constraint on future callback designs.

Remediation

Avoid `Option<AccountInfo>` for callback accounts whose valid values may include `crate::ID`.

Patch

This issue was acknowledged by the team.

© 2026 Otter Audits LLC. All Rights Reserved.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

© 2026 Otter Audits LLC. All Rights Reserved.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

© 2026 Otter Audits LLC. All Rights Reserved.